Change Propagation in Decentralized CompositeWeb Services

Walid Fdhila, Aymen Baouab, Karim Dahman, Claude Godart, Olivier Perrin, François Charoy

Change Propagation in Decentralized Composite Web Services

Walid Fdhila, Aymen Baouab, Karim Dahman,
Claude Godart, Olivier Perrin, and François Charoy
LORIA - INRIA - UMR 7503
BP 239, F-54506 Vandœuvre-lès-Nancy Cedex, France
{firstname.lastname}@loria.fr

***Abstract*—Every company wants to improve the way it does business, or produce things more efficiently, and make greater profit. Therefore, business processes have become subject to evolutionary changes, which in turn increase the need for an efficient change support. In this sense, many researches were conducted to deal with business process adaptation to changes. The latter may result in the restructuring of the whole or a part of the process. Most of the proposed approaches focus on adaptation to changes in centralized processes. In sharp contrast to these works, our operation of change adaptation that we present in this paper, concerns decentralized orchestrations. Indeed, many recent approaches were proposed to decompose a composite web service into small partitions. Since the activities, the control and data flows are distributed over these partitions, it becomes difficult to specify the changes directly. Moreover, changing a derived partition may affect the way it interacts with others. In order to overcome these deficiencies, we propose a design-time methodology to support changes in decentralized business processes. We mainly demonstrate how to propagate the changes made on a centralized specification of composite web service to its resulting decentralized sub-processes.**

***Keywords*-decentralized orchestration, business process, change propagation, web service.**

I. Introduction

The Service-Oriented Architecture (SOA) is a proven collection of principles for structuring large-scale systems in order to improve manageability and to streamline change. One of the pillars of the SOA is its ability to rapidly compose multiple services into an added-value business process, and then to expose the resulting composition as a *composite service* [1]. Despite the decentralized nature of the context of the B2B and B2C interactions, the conception and implementation of a typical business process rely on a centralized execution setting. The relevant research literature on business management confirms that the decentralization of business processes is a critical need for several reasons [1].

In a previous work, we proposed an approach to partition a composite web service [2], [3], [4]. The partitioning transforms the centralized process into behaviorally equivalent distributed sub-processes each of which related to a criteria. These partitions are executed independently at distributed locations and can be invoked remotely. They directly interact with each other using asynchronous messaging without any centralized control. The flexibility introduced by the derived decentralized processes on the other hand raises new requirements like adaptation to change. Neither our approach or similar approaches on decentralization deal with the problem of change propagation after partitioning.

However, in today's dynamic business world, the economic success of an enterprise increasingly depends on its ability to react to changes within its environment in a quick and flexible way [5]. Therefore, a critical challenge for the competitiveness of any enterprise is its ability to quickly react to business process changes and to adequately deal with them [6]. Causes for these changes can be manifold and include the introduction of new laws, market dynamics, or changes in customers' attitudes [7]. For these reasons, companies have recognized business agility as a competitive advantage, which is fundamental for being able to successfully cope with business trends.

Changes may range from simple modifications of the process to a complete restructuring of the business process to improve efficiency. In the context of the decentralized service orchestrations, applying these changes in a straightforward manner on the derived orchestration partitions is a complex maintenance task, since the control and data flows are decomposed over multiple partitions. In this sense, this paper presents a method for adaptation to change in the decentralized composite web services. In sharp contrast to previous works [8], [9], [10], [11], our change adaptation concerns decentralized orchestrations. Given a well-behaved structural update on a centralized orchestration, our approach *automates the change forward propagation* that consistently propagates the update to the derived decentralized partitions. This includes the identification of the partitions concerned by the modification, and the incorporation of the necessary changes in each of them. The main advantage of this method, is that only concerned partitions by the change are affected, and there is no need to recompute the whole decentralization or redeploy all the partitions.

The reminder of this paper is structured as follows. Section II introduces a motivating example to illustrate the importance of change adaptation in decentralized orchestrations. We adopt this example to explains the different steps of our approach. Section III presents the formal definitions needed

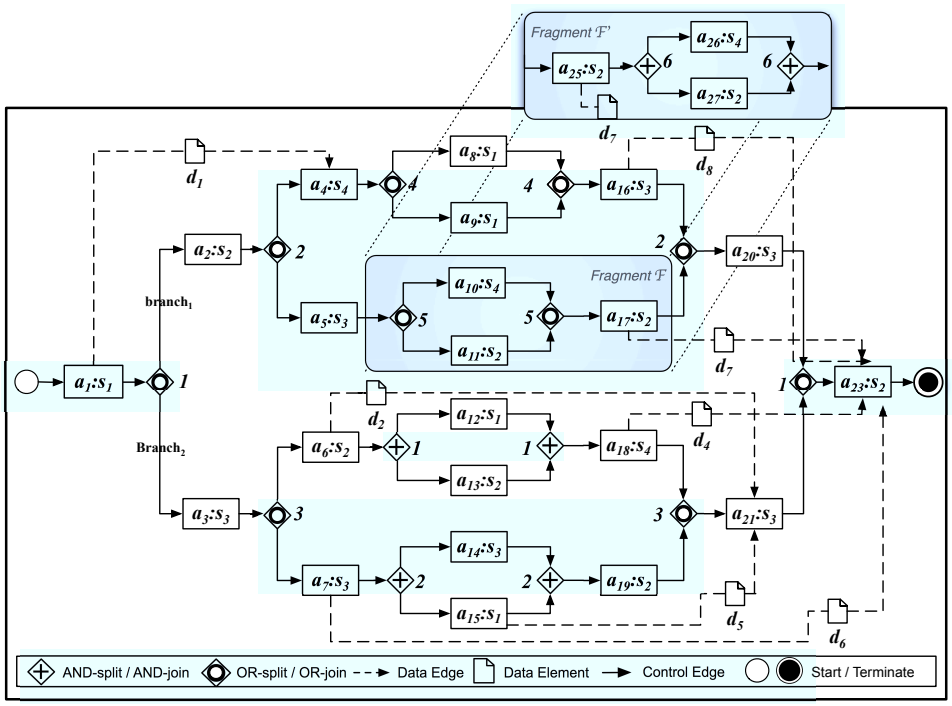

Figure 1. Process example

to provide a generic approach, while section IV details our adaptation to change mechanism for the decentralized processes. After a reviewing of the properties of our approach in Section V and its evaluation, the section VI and VII we discuss the related works, summarize the contribution and outline future directions.

II. Motivating example

The composite web services are generally captured by means of an orchestration model: a process model in which each activity represents either an intermediate work step (e.g. a data transformation) or an interaction with one of the services participating in the composition (the *component services*). The process model specifies the control-flow and data-flow relations between activities, using a specialized language such as the Business Process Execution Language (WS-BPEL) or the Business Process Modeling Notation (BPMN).

To motivate and illustrate the method presented in this paper, we make use of a sample orchestration (cf. Figure 1). The corresponding process model is captured in the BPMN notation, and it includes both control and data dependencies. The process is represented by a directed acyclic graph where nodes are activities or control patterns and edges are data or control dependencies. Activities are depicted with boxes with the activity name inside and the web service it refers to. The arcs between boxes describe the dependencies. This leads to a well defined chronological execution of the activities. In this example, we consider that services are known in advance. In particular this process represents a collaboration with four services s_1, s_2, s_3 and s_4. Data edges are represented by dashed arrows, and each control pattern has an identifier. After a_1 termination, the data d_1 is sent to a_4, then a_2 or a_3 or both of them are executed. This execution terminates when it reaches the terminate state.

Based on our partitioning techniques presented in [2], [3], Figure 2 depicts a decentralized execution settings for the process example of Figure 1. The latter is partitioned into four partitions that are executed by four distributed orchestrators (preferably collocated with the web services). Each partition P_s is responsible for all activities that are delegated to a given service and defines the relationships between them. The connectivity between activities of the centralized process is translated to that between activities of different partitions, through asynchronous message exchange mechanism. This includes data and control links (dashed arrows) represented by c_1, c_2, and d_1, d_7 respectively. The schema, illustrates only a part of the interconnection mechanism.

Now, consider that the designer wants to make some changes, which result in a restructuring of the process activities. Specifying these changes directly on the decentralized partitions is a complex and error-prone task. Indeed, the activities are distributed over partitions, and a local change on a partition may affect other partitions, since they are related. Hence, it is more easier to apply the changes

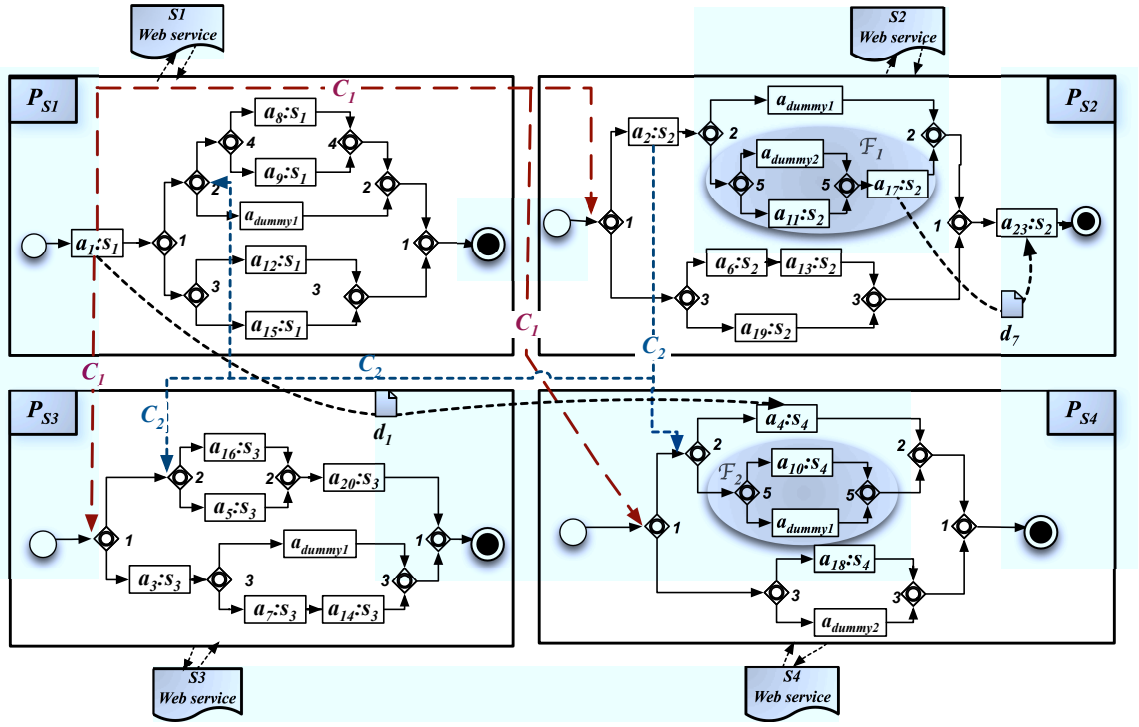

Figure 2. Decentralized partitions

on the centralized specification, and then propagate these modifications to the derived partitions with respect to control and data flows.

Let's consider the example depicted in Figure 1, and suppose that the fragment F (gray box, consisting of OR_{sp5}, a_{10}, a_{11}, a_{17} and OR_{j5}) should be replaced by the fragment $F\prime$ (consisting of AND_{sp6}, a_{25}, a_{26}, a_{27} and AND_{j5}). The data link, initially connecting a_{17} and a_{23} is modified by a data link between a_{25} and a_{23}. Obviously, this replacement would affect mainly the partitions P_{s2} and P_{s4}, since the modification includes only activities invoking s_2 and s_4 respectively (c.f. Figure 2). As can seen in the above example, the changes on the centralized specification, results in restructuring of the concerned derived partitions. They can either be done as an insertion, an update or a deletion of tasks. In Figure 2, the gray boxes in partitions P_{s2} and P_{s4}, represent the fragments which are affected by the change. These fragments are sub-processes with *single entry/single exit* and should be replaced by the updated fragments. To conclude, the main problem behind this change is: **(i)** How to identify the concerned partitions, and in each partition the fragments to be changed or restructured? **(ii)** How the concerned activities and control patterns are deleted or updated? **(iii)** Where to insert the new fragments and how? **(iv)** Finally, how to update the communications with other partitions to preserve the semantics of the initial process?

III. Formal Model

In order to provide a generic approach for change adaptation in the decentralized business processes, we adopt a high level reasoning using an abstract notation. A process is specified in an abstract way (e.g. using a graph based formalism), and mapped onto the implementation level. We do not presume any particular process modeling approach, but simply assume that the basic elements of a process can be specified in an abstract way to be translated to an executable process language (i.e. process structure in terms of atomic activities and sub-processes, dependencies between the steps of a process, etc). By definition, a process which specifies a web service composition defines the relationship between service invocations. This relationship may characterize either the control or data flow structure.

Definition 1 (Process): Formally, a process $\mathcal{P}$ is a tuple ($\mathcal{O}$, $\mathcal{D}$, $\mathcal{E}c$, $\mathcal{E}d$, $\mathcal{S}$) where

- $\mathcal{O}$ is a set of objects which can be partitioned into disjoints sets of activities $\mathcal{A}$, events ($start$ and end) and control patterns $\mathcal{CTR}$,
- $\mathcal{D}$ is a set of data,

- $\mathcal{E}c$ is a set of control edges where, $\mathcal{E}c \subset \mathcal{O}\times\mathcal{O}$.
- $\mathcal{E}d$ is a a set of data edges where, $\mathcal{E}d \subset \mathcal{A}\times\mathcal{A}\times\mathcal{D}$,
- $\mathcal{S}$ is the set of services invoked by the process.

In this paper, we consider that the processes are structured [12]. This means that different activities are structured through control elements such as AND-split, OR-split, AND-join, OR-join, etc., and for each split element, there is a corresponding join element of the same type. Additionally, the split-join pairs are properly nested. The process, has a single entry and a single exit. A process activity consists of a one-way or a bidirectional interaction with a service via the invocation of one of its operations. In conversational compositions, different operations of a service can be invoked with the execution of different activities. The set of activities that refer to the same service s is denoted $\mathcal{A}_s \mid s \in \mathcal{S}$. A control edge characterizes the mapping relationship while a data edge characterizes the mapping relation of the output and the input values of two activities.

Definition 2 (Activity): An activity $a_i \in \mathcal{A}_s$ is a tuple ($\mathcal{I}n$, $\mathcal{O}ut$, s) where $\mathcal{I}n \subset \mathcal{D}$ is the set of a_i's intputs and $\mathcal{O}ut \subset \mathcal{D}$ is the set of a_i's outputs, s is the service invoked by a_i.

Activities are related to each other and are dependent on each other. These dependencies are intra-process. Dependencies may also exist across processes and are referred to as inter-process dependencies. A path between two objects (activity, event or control pattern) is defined by the set of edges which link them. We define the $preset$ ($postset$) of an activity a_i, denoted $\bullet a_i$ ($a_i\bullet$), as the set of activities which may execute **just** before (after) a_i and related to it by a set of control dependencies.

Definition 3 (Path): The path between two objects o_i and o_j is defined by $path_{ij}$=e_1,e_2,...$e_n \in \mathcal{E}_c$ where $\forall$k, $1 \leq k < n$, target(e_k) = source(e_{k+1}) $\wedge$ source(e_1)=o_i $\wedge$ cible(e_n)=o_j.

Definition 4 (Preset): $\bullet a_i$ =$\{a_j \in \mathcal{A} | \exists\ path_{ji}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ where $\forall$k, $1<k<n$, target(e_k), source(e_k), target(e_1), source(e_n) $\in \mathcal{CTR}\}$.

Definition 5 (Postset): $a_i\bullet$ =$\{a_j \in \mathcal{A} | \exists\ path_{ij}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ where $\forall$k, $1 <k<n$, target(e_k),source(e_k), target(e_1), source(e_n) $\in \mathcal{CTR}$ $\}$.

The partitioning of a composite web service, leads to a set of interconnected partitions, each of which defines the relationship between the objects it includes. Each partition communicates with other partitions using the interaction patterns (i.e. send, receive..) [13].

Definition 6 (Partition): A sub-process or a partition is a tuple P_s = ($\mathcal{O}_s$, $\mathcal{D}_s$, $\mathcal{E}c_s$, $\mathcal{E}d_s$) where

- $\mathcal{O}_s$ is a set of objects of P_s. $\mathcal{O}_s \subset \mathcal{O} \cup \mathcal{A}_{dummy(i)}$ where $\mathcal{A}_{dummy(s)}$ is a set of dummy activities. Dummy activity is an activity with zero execution time (used for synchronization).
- $\mathcal{D}_s \subset \mathcal{D} \cup \mathcal{S}ync$, where $\mathcal{S}ync$ is a set of control data necessary for synchronization with other partitions.
- $\mathcal{E}c_s$ is the set of control edges, $\mathcal{E}c \subset \mathcal{O}_s\times\mathcal{O}_s$,
- $\mathcal{E}d_s$ is the set of data edges, $\mathcal{E}d \subset \mathcal{A}_s\times\mathcal{A}\times\mathcal{D}_s$. (control edges between partitions are transformed to data edges since they are routed in messages).
- $s \in \mathcal{S}$ is the set of services invoked by the partition.

Example: The definition of $\mathcal{P}_{s1}$ (c.f. figure 2) is the tuple $\mathcal{P}_{s1}$ = ($\mathcal{O}_{s1}$, $\mathcal{D}_{s1}$, $\mathcal{E}c_{s1}$, $\mathcal{E}d_{s1}$) where,

- $\mathcal{O}_{s1}$ = $\{a_{1,8,9,15,12}$, a_{dummy1}, $start_1$, end_1, $OR_{1,2,3,4sp}$, $OR_{1,2,3,4j}\}$.
- $\mathcal{D}_{s1}$ = $\{d_1$,d_5,..$\}$
- $\mathcal{E}c_{s1}$ = $\{e_{start,a_1}$, $e_{a_1,OR_{1sp}}$, $e_{OR_{1sp},OR_{2sp}}$, $e_{OR_{1sp},OR_{3sp}}$, ...$\}$.
- $\mathcal{E}d_{s1}$ = $\{c_1$, c_2, $e_{a_1:P_{s1},a_4:P_{s4}}(i.e.\ d_1)\}$

Next, we define the transitive *postset* (resp., transitive *preset*) of an activity a_i denoted $T\_a_i\bullet$ ($\bullet T\_a_i$), as the set of activities in the same partition as a_i, which may execute just after (before) it, and linked to it by a set of control dependencies. In the centralized process, the path between a_i and its transitive *postset* (resp., transitive *preset*) may include other activities of other partitions.

Definition 7 (Transitive postset): Formally, for an activity $a_i \in \mathcal{P}_{ck}$, is $T\_a_i\bullet$ = $\{a_j \in \mathcal{P}_{ck} | \ path_{ij}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ st $\forall\ 1<k<n$, target(e_k), source(e_k), target(e_1), source(e_n) $\notin \mathcal{A}_{ck}\}$.

Definition 8 (Transitive Preset): Formally, for an activity $a_i \in \mathcal{P}_{ck}$ $\bullet T\_a_i$ = $\{a_j \in \mathcal{P}_{ck} | \ path_{ji}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ st $\forall\ 1<k<n$, target(e_k), source(e_k), target(e_1), source(e_n) $\notin \mathcal{A}_{ck}\}$.

IV. Change propagation

In this section, we present our methodology for decentralized business processes adaptation to change. We remind that our approach concerns only already partitioned processes. This means that we do not seek to provide a change support for a centralized process. Instead, we demonstrate how to propagate the changes made on a centralized specification of a composite web service to its resulting decentralized sub-processes. The approach is structured as follows. First the designer specifies the changes using the centralized process specification, then we compute the new configurations of the decentralized fragments enclosing the changes. Finally, we propagate the changes to the concerned partitions. In this way, only the fragments which are concerned by the changes would be affected. Moreover, there is no need to re-partition the

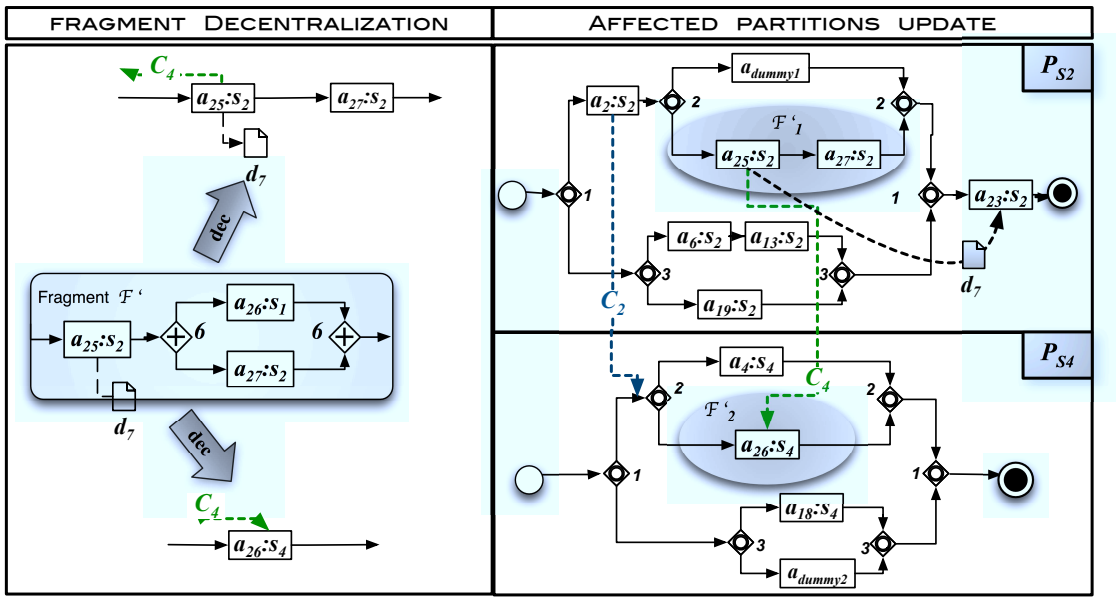

Figure 3. Change adaptation example

centralized process and re-deploy all the derived partitions.

A. Change operations

In general, process models can be decomposed into SESE fragments [14]. A SESE fragment is a non-empty subgraph in the process model with a single entry and a single exit edge [15]. For every change in the process model, there is at least one enclosing fragment. Here, we consider only the smallest fragment that encloses the changes. This can be achieved using the process structure tree (PST) [14]. In the following, we consider that the fragments enclosing the changes are already identified (the identification issue is out of scope of this paper). Formally, a fragment has the same definition as a process (c.f. definition 1), except it has no start and end events, instead it has one entry and one exit edges. We can resume the changes that can be made on a process model, by three formal operations as follows:

- ***Insert(fragment, entry, exit)***: this operation is used to insert a new fragment into the process. This fragment should be inserted between the $entry$ and $exit$ edges in the centralized process model.
- ***Delete(entry,exit)***: this operation is used for the deletion of the fragment between the $entry$ and $exit$ edges in the centralized process model.
- ***Update(fragment,entry,exit)***: this operation updates the existing fragment between $entry$ and $exit$ edges in the centralized process model, and replace it by $fragment$. This operation can also be replaced by the two consecutive operations $delete(entry, exit)$ and $insert(fragment, entry, exit)$.

It should be noted that a fragment ranges from a simple activity to a enhanced *structured* sub-process with a single entry/single exit. Let's consider the example of figure 1. The fragment $\mathcal{F}\prime$ to add in the process model is defined by $\mathcal{F}\prime = (\mathcal{O}_{\mathcal{F}\prime}, \mathcal{D}_{\mathcal{F}\prime}, \mathcal{E}c_{\mathcal{F}\prime}, \mathcal{E}d_{\mathcal{F}\prime}, \mathcal{S}_{\mathcal{F}\prime})$ and the fragment to delete is defined by $\mathcal{F} = (\mathcal{O}_{\mathcal{F}}, \mathcal{D}_{\mathcal{F}}, \mathcal{E}c_{\mathcal{F}}, \mathcal{E}d_{\mathcal{F}}, \mathcal{S}_{\mathcal{F}})$. Then, the change operation is defined by Update($\mathcal{F}\prime$, $\mathcal{F}.entry$, $\mathcal{F}.exit$) where $\mathcal{F}.entry = e_{a_5,OR_{5sp}}$ and $\mathcal{F}.exit = e_{a_{17},OR_{5j}} \in \mathcal{E}c_B$. This operation can also be transformed as follows:

> Update($\mathcal{F}\prime$, $\mathcal{F}.entry$, $\mathcal{F}.exit$) $\Rightarrow$ Delete($\mathcal{F}.entry$, $\mathcal{F}.exit$) $\wedge$ Insert($\mathcal{F}\prime$, $\mathcal{F}.entry$, $\mathcal{F}.exit$).

B. Change adaptation

This section describes the different steps to propagate the changes made on the centralized process model to the derived decentralized partitions. To have a better understanding, we refer to the motivating example depicted in Figure 1) to illustrate the adaptation process, then we detail our approach using a generic process model.
We remind that we want to replace the fragment $\mathcal{F}$ in the centralized process by the fragment $\mathcal{F}\prime$. For this purpose, we first identify the partitions affected by this change using activities identifiers. For instance, the deletion of activity $a_{10} : s_4$ in $\mathcal{F}$ or the insertion of the new activity $a_{25} : s_2$ in $\mathcal{F}\prime$ means the deletion or the insertion of the same activities from P_{s4} or to P_{s2} respectively. The second step,

is to identify the blocs of activities to change inside the affected partitions. Indeed, a simple change in a partition may result in other modifications (including interactions with other partitions or control patterns). For example (c.f. figure 2), the deletion of the activity a_{11} in P_{s2} results in the deletion of all the fragment enclosed between OR_{5sp} and OR_{5j} since the other activity in parallel with it, is dummy. For each identified fragment, we notice the entry and the exit edges. Two use cases are possible: the fragment is updated and replaced by another fragment, or deleted. Using the new fragment $\mathcal{F}\prime$, the next step is to determine what to insert in each partition. For this purpose, we have to partition $\mathcal{F}\prime$.

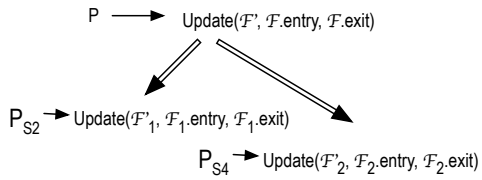

Figure 4. Operation transformation

As can seen in Figure 3, the partitioning of the fragment $\mathcal{F}\prime$, leads to two fragments $\mathcal{F}\prime_1$ and $\mathcal{F}\prime_2$, connected by a control edge c_4 (for more details about the decentralization process refer to [2]). The affected fragments by the change are respectively, $\mathcal{F}_1$ and $\mathcal{F}_2$, and therefore, the initial change operation is transformed into two updates on P_{s2} and P_{s4} respectively. Figure 4 illustrates the transformation operation, where $\mathcal{F}_1$ is updated by $\mathcal{F}\prime_1$ in P_{s2} and $\mathcal{F}_2$ is updated by $\mathcal{F}\prime_2$ in P_{s4}.

Now, consider a general example of a centralized process model $\mathcal{P}$, as depicted in Figure 5. The process model is structured through split and join patterns, and enclosed with a start / end events. The partitioning of this process, results in $|S|$ interconnected partitions P_I, P_J, P_K, etc, each of which executed by a separate orchestrator (c.f. Figure 6). The steps toward change propagation are as follows:

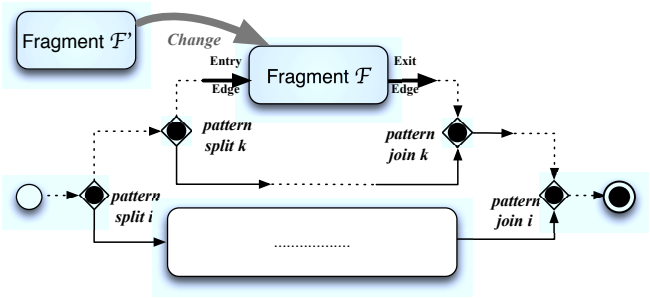

Figure 5. generic process example

1- Change specification: the designer specifies the changes to do using change operations: Insert, Delete and Update. If the operation is a delete, then he has to indicate the concerned fragment (i.e. in figure 5 $Delete(Entry_{edge}, Exit_{edge})$). If the operation is an Insert, then he has to specify the fragment to add and in which place in the process model (i.e. in figure 5 $Insert(\mathcal{F}\prime, Entry_{edge}, Exit_{edge})$). Otherwise, he has to specify both the fragment to update and the new fragment to insert (i.e. in Figure 5 $Update(\mathcal{F}\prime, \mathcal{F}.entry, \mathcal{F}.exit)$). Next, we consider the Update operation since it is more general and includes the two other operations.

2- Partitions identification: Using the fragments $\mathcal{F}$ and $\mathcal{F}\prime$, we identify all the partitions that would be affected by the change. Indeed, during partitioning, each activity is assigned to a partition upon to a certain criteria. If the activity responds to the criteria of the partition then it would be assigned to it (i.e. activities having the same role, or invoking the same service). So, using the criteria assigned to each activity we can determine the partition it would belong to. By this way, each partition having a criteria of one of $\mathcal{F}$ or $\mathcal{F}\prime$ activities would be affected by the change.

3- Fragments partitioning: this step consists in decentralizing separately the fragments $\mathcal{F}\prime$ and $\mathcal{F}$ into interconnected sub-fragments, using partitioning techniques for structured processes. In figure 6, we take into consideration only the sub-fragments $\mathcal{F}_{PJ}\prime$, $\mathcal{F}_{PK}\prime$, $\mathcal{F}_{PI}$ and $\mathcal{F}_{PJ}$, since they cover the three possible scenarios: insert, delete or update a sub-fragment into a partition.

4- Change translation: After $\mathcal{F}$ and $\mathcal{F}\prime$ partitioning, change operation for the process model is decomposed into one or more change operations. Each operation represents the change to make on the corresponding partition. The generic formula for operation transformation is $operation(x, y, ..) \Rightarrow operation_1(x_1, y_1, ..) \wedge ... \wedge operation_2(x_2, y_2, ...)$, where $operation_i$ is the change to apply to partition P_i. For instance, the generated change operations on partitions P_I, P_J and P_K are as follows:

$$Update_P(\mathcal{F}\prime,\ \mathcal{F}.entry,\ \mathcal{F}.exit) \Rightarrow Update_{PJ}(\mathcal{F}\prime_{PJ},\ \mathcal{F}_{PJ}.entry,\ \mathcal{F}_{PJ}.exit) \wedge Delete_{PI}(\mathcal{F}_{PI}.entry,\ \mathcal{F}_{PI}.exit) \wedge Insert_{PK}(\mathcal{F}\prime_{PK},\ entry,\ exit).$$

It should be noticed, that the entry and the exit of the $Insert$ construct should be computed (next step).

5- Partitions adaptation to change: This step consists in applying the changes to the corresponding partitions. For this purpose, we first, have to determine exactly where to insert the sub-fragments $\mathcal{F}_{PJ}\prime$, $\mathcal{F}_{PK}\prime$. The first scenario related to the update of $\mathcal{F}_{PJ}\prime$ in P_J is simple, since we

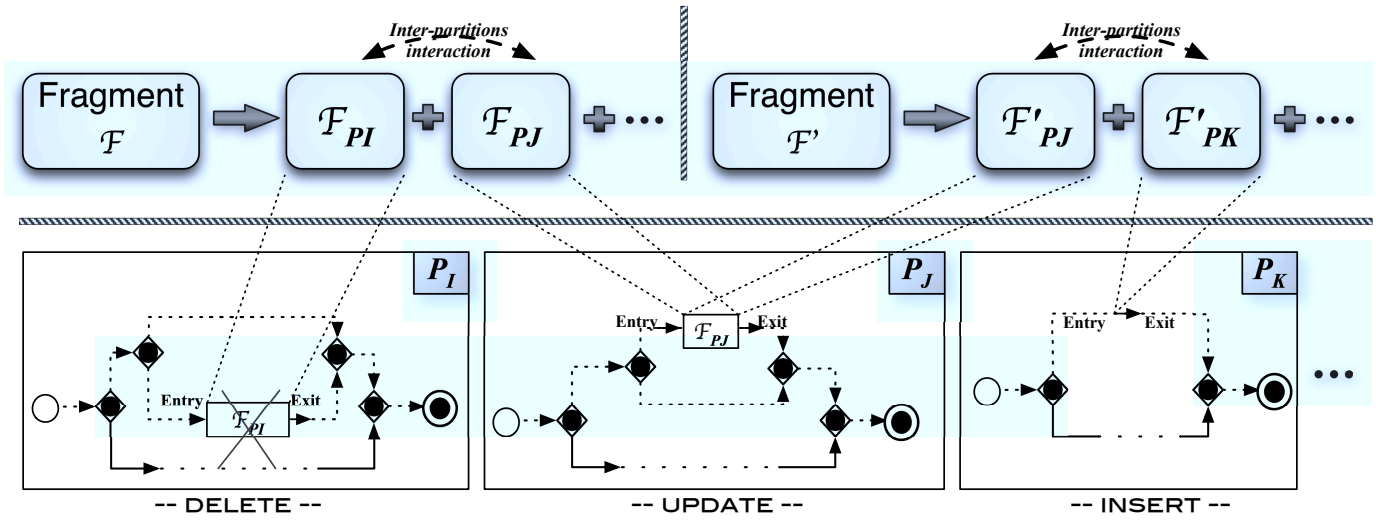

Figure 6. generic example for change management

already know the entry and exit edges of $\mathcal{F}_{PJ}$. So, we have just to look for these edges in the partition and replace all the fragment between them by the fragment $\mathcal{F}_{PJ}\prime$. The deletion of the latter, implies the deletion of all the interactions with other activities in the same partition or other partitions. The partitions which interact with any activity concerned by the change is also concerned by the change, since we have to update its corresponding interaction edges. Formally, the update of $\mathcal{F}_{PJ}$ in a P_J by $\mathcal{F}_{PJ}\prime$ corresponds to the deletion of all objects $o \in \mathcal{O}_{\mathcal{F}_{PJ}}$, edges $e \in \mathcal{E}c_{\mathcal{F}_{PJ}} \cup \mathcal{E}d_{\mathcal{F}_{PJ}}$, and data, and their substitution by the objects, edges, and data of $\mathcal{F}\prime_{PJ}$.

The Delete operation application is similar to the update, except that we do not insert a new sub-fragment. We simply look for the entry and exit of $\mathcal{F}_{PI}$ in the partition P_I. Then we delete the sub-fragment between them. If the entry edge of the sub-fragment to delete, is linked to a (choice or parallel) split control patterns (outside the sub-fragment), and the exit edge is linked to its corresponding join element, then we look if the other branches linking these two elements include only dummy activities or not. If yes then we delete these two patterns. We iterate this operation on each nested constructs linking the sub-fragment $\mathcal{F}_{PI}$ to its transitive $preset$ $\bullet T_{\mathcal{F}_{PI}}$ and $postset$ $T_{\mathcal{F}_{PI}}\bullet$ (c.f. definition 7, 8) (we extend the definition of transitive postset (transitive preset) to that between a fragment and its subsequent (previous) activities). Otherwise, we replace the sub-fragment to delete, by a dummy activity.

Now, to insert sub-fragment $\mathcal{F}_K\prime$ in the partition P_K, we have to identify the $entry$ and $exit$ edges. For this purpose, we first compute the transitive preset and postset of $\mathcal{F}_K\prime$ in P_K ($\bullet\mathcal{F}_K\prime$, $\mathcal{F}_K\prime\bullet$). Then, we identify all the control patterns that link them in the centralized process model. Next, we identify each split pattern ctr in this control path linking it to its $\mathcal{F}_K\prime\bullet$, such as $\overline{ctr}$ is in the path linking it to its $\bullet\mathcal{F}\prime_K$ ($\overline{ctr}$ is the correspondent join element of ctr). For each *choice* ctr found, we look if it already exists in the partition. If yes, we just add a new branch linking ctr to $\overline{ctr}$ in which we put $\mathcal{F}\prime_K$. If no, we add it and its corresponding $\overline{ctr}$, then we put the $\mathcal{F}\prime_K$ between them (in parallel with a dummy activity). In some cases, the update or the insertion of a fragment may result in the creation of a new partition or the deletion of an existing partition.

The propagation of the changes made on the centralized process to the derived decentralized partitions are formally described by three rules. These rules include all the steps mentioned previously. According to the change operation (update, delete or insert), we execute the corresponding actions. These actions represent the identification of the affected partitions by the change, the identification of the fragments to be changed in these partitions and the modifications to apply. The modifications range from the change operations to apply on the concerned partitions to the update of the interactions with other partitions. In these rules $dec(\mathcal{F})$ is a function that returns the decentralized sub-fragments of the fragment $\mathcal{F}$.

Rule 1
Operation: $Delete_p(\mathcal{F}.entry, \mathcal{F}.exit)$, $\mathcal{F} \subset p$
Actions:
$\forall\ \mathcal{F}_i \in dec(\mathcal{F})$, $p_i \leftarrow IdentifyPartitionOf(\mathcal{F}_i)$
$\forall\ \mathcal{F}_i \in dec(\mathcal{F})$, $(entry_i, exit_i) \leftarrow IdentifyIn(\mathcal{F}_i, p_i)$
$\forall\ \mathcal{F}_i \in dec(\mathcal{F})$, $Delete_{p_i}(entry_i, exit_i)$
$\forall a_j \in \bullet\mathcal{F}$, $\forall a_k \in \mathcal{F}\bullet$ $update\_connection(a_j, a_k)$

Rule 2
Operations: $Insert_p(\mathcal{F}\prime, entry, exit)$
Actions:
$\forall\ \mathcal{F}_i\prime \in dec(\mathcal{F}\prime)$, $p_i \leftarrow IdentifyPartitionOf(\mathcal{F}\prime_i)$
$\forall\ \mathcal{F}_i\prime \in dec(\mathcal{F}\prime)$, $(entry_i, exit_i) \leftarrow IdentifyIn(\mathcal{F}_i\prime, p_i)$
$\forall\ \mathcal{F}_i\prime \in dec(\mathcal{F}\prime)$, $Insert_{p_i}(\mathcal{F}_i\prime, entry_i, exit_i)$
$\forall a_j \in \bullet\mathcal{F}\prime$, $update\_connection(a_j, \mathcal{F}\prime)$
$\forall a_k \in \mathcal{F}\prime\bullet$ $update\_connection(\mathcal{F}\prime, a_k)$

Rule 3
Operations: $Update_p(\mathcal{F}\prime, \mathcal{F}.entry, \mathcal{F}.exit)$, $\mathcal{F} \subset p$
Actions:
$\forall \mathcal{F}_i\prime \in dec(\mathcal{F}\prime)$, $p_i \leftarrow IdentifyPartitionOf(\mathcal{F}\prime_i)$
$\forall \mathcal{F}_j \in dec(\mathcal{F})$, $p_j \leftarrow IdentifyPartitionOf(\mathcal{F}_j)$
$\forall\ \mathcal{F}_i \in dec(\mathcal{F})$, $(entry_i, exit_i) \leftarrow IdentifyIn(\mathcal{F}_i, p_i)$
$\forall \mathcal{F}_i \in dec(\mathcal{F})$, $\forall \mathcal{F}_j\prime \in dec(\mathcal{F}\prime)$ s.t. $p_i{=}p_j$, $Update_{p_i}(\mathcal{F}_j\prime, entry_i, exit_i)$
$\forall \mathcal{F}_j\prime \in dec(\mathcal{F}\prime)$ s.t. $\nexists \mathcal{F}_i \in dec(\mathcal{F}) \wedge p_i \neq p_j$, $Insert_{p_j}(\mathcal{F}_j\prime, entry_j, exit_j)$ where $(entry_j, exit_j) \leftarrow IdentifyIn(\mathcal{F}_j\prime, p_j)$
$\forall \mathcal{F}_i \in dec(\mathcal{F})$ s.t. $\nexists \mathcal{F}_j\prime \in dec(\mathcal{F}\prime) \wedge p_i \neq p_j$, $Delete_{p_j}(\mathcal{F}_i, entry_i, exit_i)$ where $(entry_i, exit_i) \leftarrow IdentifyIn(\mathcal{F}_j, p_i)$
$\forall a_j \in \bullet\mathcal{F}$, $update\_connection(a_j, \mathcal{F}\prime)$
$\forall a_k \in \mathcal{F}\bullet$ $update\_connection(\mathcal{F}\prime, a_k)$

V. Evaluation of the approach

This section presents the properties of our change propagator. Given a well-behaved structural update on the centralized process model and the derived decentralized subprocesses, our approach *automates the change forward propagation* that consistently transforms the update on the source into the related target partitions, as presented in Section IV. Then we show the applicability of our approach by reviewing the motivating example and presenting an implementation.

A. The properties of the approach

In order to describe this approach following our work in [16] and defined in definitions 1 and 6, the *process partitioning* is a total function of the type $dec : \mathcal{P} \rightarrow \{\mathcal{P}_s\}^{s\in\mathcal{S}}$ that takes a source centralized process in $\mathcal{P} = (\mathcal{O}, \mathcal{D}, \mathcal{E}c, \mathcal{E}d, \mathcal{S})$ and produces a target set of decentralized partitions $\mathcal{P}_s = (\mathcal{O}_s, \mathcal{D}_s, \mathcal{E}c_s, \mathcal{E}d_s)$ where $s \in \mathcal{S}$. It establishes a *consistency relation*, denoted $\mathcal{C} \subseteq \mathcal{P} \times \{\mathcal{P}_s\}^{s\in\mathcal{S}}$ between the source and the target process models. This relation captures the mapping between the centralized process logic in $\mathcal{P}$ and the (same) decentralized process logic described in $\{\mathcal{P}_s\}s \in \mathcal{S}$. Since our decentralization algorithm is idempotent as we have presented in [2]: it can be applied multiple times without changing the result, then $\mathcal{C}$ is a total function.

Now, consider a source centralized process model $p \in \mathcal{P}$ and a target partition set $dec(p)$ that are consistent with respect to the relation $\mathcal{C}$, i.e., $(p, dec(p)) \in \mathcal{C}$ means that p was previously decentralized to $dec(p)$. Given a source change $\delta_\mathcal{P}$ that alters p to $\delta_\mathcal{P}(p)$, *the problem is to translate the well-behaved change of the source process $\delta_\mathcal{P}$ into a well-behaved changes on the target partitions* $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}}$, such that the application of both updates results in consistent process models. The change propagator that provides this function is of the type $prop : \mathcal{P} \times \Delta_\mathcal{P} \times \{\mathcal{P}_s\}^{s\in\mathcal{S}} \rightarrow \{\mathcal{P}_s\}^{s\in\mathcal{S}}$. For $p \in \mathcal{P}$, $\delta_\mathcal{P} \in \Delta_\mathcal{P}$ and $dec(p) \in \{\mathcal{P}_s\}^{s\in\mathcal{S}}$, it determines $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}} \in \Delta_{\mathcal{P}_\mathcal{S}}$ and then computes the changes on the partitions $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}}$ such that the updated models are consistent, i.e., $(\delta_\mathcal{P}(p), (\delta_{\mathcal{P}_s}(dec(p)))^{s\in\mathcal{S}}) \in \mathcal{C}$. We use $\Delta_\mathcal{P} : \mathcal{P} \rightharpoonup \mathcal{P}$ and $\Delta_{\mathcal{P}_\mathcal{S}} : \{\mathcal{P}_s\}s \in \mathcal{S} \rightharpoonup \{\mathcal{P}_s\}^{s\in\mathcal{S}}$ as an abbreviation for the update types respectively on the processes and on the partitions. They represent the space of all partial functions describing the changes on each of the centralized and decentralized process models and which can be described by productions, i.e. the change operation defined in Section IV.

In our semantics, a process and its decentralization result (i.e., the derived partitions) are specified with graphs as introduced in Section II. Then, a change on a process implies a modification on the graph structure which can be expressed by *graph rewriting* rules [17]. Formally, given a graph $\mathcal{G}$, a graph rewriting rule (i.e., also called production) consists of injective morphisms of the form $\delta_\mathcal{G} : \mathcal{L} \rightarrow \mathcal{R}$ that transform a source graph $\mathcal{L}$ into a target graph $\mathcal{R}$. In order to apply this rewrite rule to the initial graph $\mathcal{G}$, a match $m : \mathcal{L} \rightarrow \mathcal{G}$ is needed to specify which part of $\mathcal{G}$ is being updated. Then, the application of $\delta_\mathcal{G}$ to $\mathcal{G}$ via a match m for $\delta_\mathcal{G}$ is uniquely defined by the graph rewriting $\mathcal{G} \Rightarrow_{\delta_\mathcal{G}, m} \mathcal{H}$. This rule application induces a co-match $m' : \mathcal{R} \rightarrow \mathcal{H}$ which specifies the embedding of $\mathcal{R}$ in the result graph $\mathcal{H}$. For more details on the so called *forward satisfaction* associated to the graph rewriting refer to [17]. Generally, the problem of a graph rewriting technique is that it is considered to be undecidable. However, if its meets certain criteria, then, we can conclude that it is terminating and locally confluent.

Firstly, the most important criteria is the change propagation correctness: a graph-based change propagator must return consistent process models. In this paper, we suppose that when applying a rewriting rule to a given graph $\mathcal{G}$, it is enough to consider the case where the morphisms that matches $\mathcal{L}$ to $\mathcal{G}$ is injective [18], and that the match m is a total label-preserving, type-preserving and root-preserving [17] graph morphism. However, to be correctly applied, the productions must satisfy the structural consistency of the centralized process constraints. Note that we assume the well-behavedeness of the updates propagated by the designers. It means that the graph production on a centralized process is consistent with the behavioral requirements, and after the production the process remains structured. Moreover, the fragment or process partitioning preserves by definition the well-behaved process semantic. Secondly, a fundamental law is that the change propagation should be

deterministic [19]: for each centralized process model input there is a unique decentralization result. In our case, the change propagator is modeled by a mathematical function. It means that given the same pair of the centralized and its decentralized models, and a finite set of changes (i.e., bounded within the SESE fragment) on the source centralized process model, our propagator produces the same changes on the target partitions.

Finally, our propagator takes as a parameters the previous decentralization, i.e., $(p, dec(p)) \in \mathcal{C}$, and the update of the source process $\delta_{\mathcal{P}}$. In order to adapt $dec(p)$ the potential changes induced by $\delta_{\mathcal{P}}$, and rather than executing the decentralization afresh on the entire updated centralized process model, i.e., $dec(\delta_{\mathcal{P}}(p))$, our change propagator enforces an in-place synchronization between $\delta_{\mathcal{P}}(p)$ and $dec(p)$ by translating the updates $\delta_{\mathcal{P}}$ into well-behaved target updates $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}}$ to get $(\delta_{\mathcal{P}_s}(dec(p)))^{s\in\mathcal{S}})$ consistent with $\delta_{\mathcal{P}}(p)$. The *change translation* can be seen as a partial function of the type $\Delta_{\mathcal{P}} \rightharpoonup \Delta_{\mathcal{T}}$. The reason is that in the case where $\delta_{\mathcal{P}}$ is small, generally, it corresponds to a small update of the partitions: $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}}$, and the performance savings for those computations are expected to be high in comparison with a *diff*-based method [20], [17]. Taking into account a fragment in the source process p and a fragment in the previously obtained partitions $dec(p)$, the effort to determine $(\delta_{\mathcal{P}_s})^{s\in\mathcal{S}}$ with the *partition identification* and to compute $(\delta_{\mathcal{P}_s}(dec(p)))^{s\in\mathcal{S}}$ should be much less than to apply a new decentralization $dec(\delta_{\mathcal{P}}(p))$ and a *diff* between the new result and the previous $dec(p)$. The speedup of this incremental decentralization and the partitions adaptation to change results in a reasonable decoupling from the processes and partitions size [21].

B. Proof of concepts prototype

The *change propagator* has been successfully implemented and integrated with our previous development of the *partitioning algorithm* [2] as an extension to a BPMN Editor [22]. This BPMN editor is based on a graph visualization library, and it is used to model a source centralized process model, for instance the structured process of Figure 1. After applying our *partitioning algorithm*, we obtain the partitions depicted in Figure 2 using the graph library. Moreover, we have developed a filter that logs the process model editing operations presented in Section IV. Actually, the specification of the entry and the exit of a fragment is performed manually, for example as depicted in Figure 3, but it can be easily automated. The *change propagation algorithm* is implemented in the *DROOLS* [23] inference engine, and it automatically computes the graph editing operation sequence that manipulates the partitions. The experimental results are encouraging, but still needs to be validated in a real-scale case. We refer the reader to [21] for further discussions on the scalability measurements of a similar incremental model synchronization technique.

VI. RELATED WORK

Several issues related to change management have been addressed in business process management, and workflow literature [24], [25], [26]. These proposals deal with adaptive process management. For instance, the ADEPT proposal enables controlled changes at the process type as well as the process instance level [9]. The authors, present a formal foundation for the support of dynamic structural changes of running workflow WF instances. Based upon a formal WF model (ADEPT). They also define a set of change operations (ADEPTflex) that support users in modifying the structure of a running WF.

In [8], Van der Aalst and Jablonski present important issues related to process changes and discuss organizational structures. In [11], the authors motivate the need for the controlled change of organizational models. In particular, they present different adaptations models to be supported by respective components (e.g. to extend, reduce, replace, and re-link model elements).

Alanen and Porres describe in [20] an algorithm to compute elementary change operations. In [27], Kolovos et al. describe the Epsilon merging language. The latter is used to specify how models are merged. Kelter et al. present in [28] a generic model differencing algorithm. In [29], Cicchetti et al. propose a metamodel for the specification and detection of syntactical and semantical conflicts. Rinderle et al. [14] have studied disjoint and overlapping process model changes in the context of the problem of migrating process instances.

All the mentioned approaches, address change adaptation in a centralized process. They also deal, with how to dynamically adapt running instances to changes. This, may be complementary to our work.

In the decentralized setting, [14] presents a formal model for a distributed workflow change management (DWFCM) that uses a rules topic ontology and a service ontology to support the needed run-time flexibilty. A system architecture and the workflow adaptation process are presented. The approach aims to generate a new workflow that is migration consistent with the original workflow. This work is different from our proposal, since they do not seek to propagate a pre-defined changes on a centralized process to that on the derived partitions. Their work is more focused on run-time adaptation using the migration rules.

In [16] the authors present a unidirectional model incremental transformation approach. Its central contribution is the definition and the realization of an automatic synchronizer for managing and re-establishing the structural consistency of heterogeneous source and target models. From a design-time perspective, an evolution describes the update on model's internal structures that can be assimilated to a graph. They express the model evolutions and their transformation using conditional graph rewriting techniques.

VII. Conclusion

Lifecycle support of BPM solutions is becoming more and more important. In order to achieve the goals of BPM implementation, it is required to support process analysis and improvement, which necessitates a framework for managing business process changes. In this paper, we have presented an approach to adapt decentralized orchestrations to changes specified on the corresponding centralized process. The proposed approach is based on three change patterns $Insert$, $Update$ and $Delete$. The method consists in partitioning the fragment to change into sub-fragments, which in turn, are integrated into the corresponding partitions. To the best of our knowledge, this is the first work that takes on changes adaptation in decentralized composite web services. Evidently, there are some limitations inherently implied by the usage of only the structured processes. Thus, as a perspective we further to extend our approach to handle non-structured processes. Moreover, the introduced operations can be composed to give rise to process update types with enhanced semantics for the users (e.g., move of fragment, refactoring of fragments: splitting and merging). However, we restricted our presentation to the structural correctness. The integration of our update translation operator in a previously developed *ATLAS Transformation Language* chain [30] for the *Eclipse SOA Tools Platform* is in development.

References

[1] B. Benatallah, Q. Z. Sheng, and M. Dumas, "The self-serv environment for web services composition," *IEEE Internet Computing*, vol. 7, no. 1, pp. 40–48, 2003.

[2] W. Fdhila, U. Yildiz, and C. Godart, "A flexible approach for automatic process decentralization using dependency tables," in *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*. Los Angeles, CA, USA: IEEE Computer Society, 2009, pp. 847–855.

[3] W. Fdhila and C. Godart, "Toward synchronization between decentralized orchestrations of composite web services," in *CollaborateCom 2009, 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 11-14 2009, pp. 1 –10.

[4] W. Fdhila, M. Dumas, and C. Godart, "Optimized decentralization of composite web services," in *CollaborateCom 2010, 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 11-14 2010, pp. 1 –10.

[5] M. Hammer and S. A. Stanton, *The reengineering revolution: A handbook*. New York: HarperBusiness, 1995.

[6] W. M. P. van der Aalst, "Generic workflow models: How to handle dynamic change and capture management information?" in *CoopIS*, 1999, pp. 115–126.

[7] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems - a survey," *Data Knowl. Eng.*, vol. 50, no. 1, pp. 9–34, 2004.

[8] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in *OTM Conferences (1)*, 2007, pp. 77–94.

[9] M. Reichert and P. Dadam, "Adept$_{\text{flex}}$-supporting dynamic changes of workflows without losing control," *J. Intell. Inf. Syst.*, vol. 10, no. 2, pp. 93–129, 1998.

[10] M. Adams, A. H. M. ter Hofstede, W. M. P. van der Aalst, and D. Edmond, "Dynamic, extensible and context-aware exception handling for workflows," in *OTM Conferences (1)*, 2007, pp. 95–112.

[11] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change patterns and change support features - enhancing flexibility in process-aware information systems," *Data Knowl. Eng.*, vol. 66, no. 3, pp. 438–466, 2008.

[12] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler, "On structured workflow modelling," in *CAiSE*, 2000, pp. 431–445.

[13] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede, "Service interaction patterns," in *Business Process Management*, 2005, pp. 302–318.

[14] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," in *ICSOC*, 2007, pp. 43–55.

[15] J. M. Küster, C. Gerth, and G. Engels, "Dynamic computation of change operations in version management of business process models," in *ECMFA*, 2010, pp. 201–216.

[16] K. Dahman, F. Charoy, and C. Godart, "Towards consistency management for a business-driven development of soa," in *The 15th IEEE International Enterprise Distributed Object Computing Conference*, Helsinki, Finland, 2011.

[17] E. Biermann, C. Ermel, and G. Taentzer, "Precise Semantics of EMF Model Transformations by Graph Transformation," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, ser. MoDELS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 53–67.

[18] M. Antkiewicz and K. Czarnecki, "Design space of heterogeneous synchronization," in *Generative and Transformational Techniques in Software Engineering II*, R. Lämmel and J. Visser, Eds. Berlin, Heidelberg: Springer, 2008, pp. 3 – 46.

[19] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *The 22th IEEE international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 164–173.

[20] M. Alanen and I. Porres, "Difference and union of models," in *UML*, 2003, pp. 2–17.

[21] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software and Systems Modeling*, vol. 8, pp. 21–43, 2009.

[22] "Yaoqiang bpmn editor," (Jul. 2011). [Online]. Available: bpmn.yaoqiang.org

[23] "Drools," (Feb. 2011). [Online]. Available: labs.jboss.com/drools/

[24] W. M. P. van der Aalst, "Exterminating the dynamic change bug: A concrete approach to support workflow change," *Information Systems Frontiers*, vol. 3, no. 3, pp. 297–317, 2001.

[25] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems - a survey," *Data Knowl. Eng.*, vol. 50, no. 1, pp. 9–34, 2004.

[26] W. Sadiq, O. Marjanovic, and M. E. Orlowska, “Managing change and time in dynamic workflow processes,” *Int. J. Cooperative Inf. Syst.*, vol. 9, no. 1-2, pp. 93–116, 2000.

[27] D. S. Kolovos, R. F. Paige, and F. Polack, “Merging models with the epsilon merging language (eml),” in *MoDELS*, 2006, pp. 215–229.

[28] U. Kelter, J. Wehren, and J. Niere, “A generic difference algorithm for uml models,” in *Software Engineering*, 2005, pp. 105–116.

[29] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, “Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach,” in *Handbook of Graph Grammars*, 1997, pp. 163–246.

[30] K. Dahman, F. Charoy, and C. Godart, “Generation of Component Based Architecture from Business Processes: Model Driven Engineering for SOA,” in *The 8th IEEE European Conference on Web Services*, Ayia Napa, Greece, 2010.